SIAM J. SCI. COMPUT.
Vol. 37, No. 3, pp. C415–C438

COMPUTING THE GRADIENT IN OPTIMIZATION ALGORITHMS FOR THE CP DECOMPOSITION IN CONSTANT MEMORY THROUGH TENSOR BLOCKING*

NICK VANNIEUWENHOVEN†, KARL MEERBERGEN†, AND RAF VANDEBRIL†

Abstract. The construction of the gradient of the objective function in gradient-based optimization algorithms for computing an r-term CANDECOMP/PARAFAC (CP) decomposition of an unstructured dense tensor is a key computational kernel. The best technique for efficiently implementing this operation has a memory consumption that scales linearly with the number of terms r and sublinearly with the number of elements of the tensor. We consider a blockwise computation of the CP gradient, reducing the memory requirements to a constant. This reduction is achieved by a novel technique that we call implicit block unfoldings, which combines the benefits of the block tensor unfoldings by [Ragnarsson and Van Loan, *SIAM J. Matrix Anal. Appl.*, 33 (2012), pp. 149–169] and the implicit unfoldings of [Phan, Tichavský, and Cichocki, *IEEE Trans. Signal Process.*, 61 (2013), pp. 4834–4846]. A heuristic algorithm for automatically choosing the division into subtensors is part of the proposed algorithm. The throughput that can be attained is essentially determined by the performance of a matrix product of two small matrices of constant size. Numerical experiments illustrate that the proposed method can outperform the current state-of-the-art by up to two orders of magnitude for large dense tensors in terms of memory consumption, while the increase of the execution time is no more than 5%. The proposed algorithm attained upward of 90% of the theoretical peak performance of the computer system, using no more than 50MB of memory, irrespective of the size of the tensor and the number of terms r.

Key words. CANDECOMP/PARAFAC, tensor rank decomposition, CP decomposition, CP gradient, implicit block unfolding

AMS subject classifications. 15A69, 65Y20, 68W40, 65K10, 65F30

DOI. 10.1137/14097968X

1. Introduction.

Multidimensional data can be organized as a d-array of numbers

$$\mathcal{A} = [a_{i_1,i_2,\ldots,i_d}]_{i_1,i_2,\ldots,i_d=1}^{n_1,n_2,\ldots,n_d} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d},$$

which we will refer to as a tensor. Such data arise naturally in several applications and are ubiquitous in engineering and science. Consider, for example, the application discussed in [27] from aerospace engineering where numerical simulations of the airflow around an airfoil are a cost-effective procedure for design prototyping prior to wind tunnel testing. In the simplest setting, such a computational fluid dynamics simulation

*Submitted to the journal's Software and High-Performance Computing section July 28, 2014; accepted for publication (in revised form) April 17, 2015; published electronically June 18, 2015.
http://www.siam.org/journals/sisc/37-3/97968.html

†Numerical Approximation and Linear Algebra Group, Department of Computer Science, KU Leuven, 3000 Leuven, Belgium (nick.vannieuwenhoven@cs.kuleuven.be, karl.meerbergen@cs.kuleuven.be, raf.vandebril@cs.kuleuven.be). The first author acknowledges the support of a Ph.D. Fellowship of the Research Foundation – Flanders (FWO). The second and third authors acknowledge support by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office, Belgian Network DYSCO (Dynamical Systems, Control, and Optimization). The second author additionally acknowledges the support of KU Leuven Research Council grants PFV/10/002 (Optimization in Engineering) and OT/10/038 (Multi-parameter Model Order Reduction and Its Applications), while the third author was additionally supported by the Research Foundation – Flanders (FWO) project G034212N (Reestablishing Smoothness for Matrix Manifold Optimization via Resolution of Singularities), the KU Leuven Research Council project OT/11/055 (Spectral Properties of Perturbed Normal Matrices and Their Applications), and CREA/13/012 (Can Unconventional Eigenvalue Algorithms Supersede the State-of-the-Art).

yields three spatial coordinates, three velocity components, the pressure, and the temperature at every discretization point. These simulations are performed for various configurations of the airfoil that may be encountered in the course of flight; this leads to additional dimensions corresponding to the angle of attack, yaw angle, the Reynolds number, and the Mach number. Such simulations result in huge amounts of data, leading to a desire to compress the data without losing information. In this regard, tensor decompositions are a natural approach that explicitly cater to the data's multidimensional nature.

Hitchcock [22, 23] proposed the following decomposition of a tensor $\mathcal{A}$:

$$\mathcal{A} = \sum_{i=1}^{r} \alpha_i \mathbf{v}_1^{(i)} \otimes \mathbf{v}_2^{(i)} \otimes \cdots \otimes \mathbf{v}_d^{(i)},$$

where $\mathbf{v}_k^{(i)} \in \mathbb{R}^{n_k}$ for every $k = 1, \ldots, d$ and $i = 1, \ldots, r$, $\alpha_i \in \mathbb{R}$, and $\otimes$ is the tensor product. We refer to the above decomposition as an r-term CANDECOMP/PARAFAC (CP) decomposition [8, 20]. In this expression, r is called the rank of $\mathcal{A}$ if no such expression of the above type exists with a strictly smaller number of terms. A key feature of this decomposition, which is often considered its prime advantage over matrix decompositions, is its uniqueness; see, e.g., [7, 9, 12] for contemporary results. An early prototypical application of this decomposition in the context of data analysis is found in chemometrics, where Appellof and Davidson [5] elucidated that the fluorescence intensity of a pure fluorophore measured at a discrete time t_k emitting light at wavelength m_i when excited with light of wavelength x_k can be modeled as a rank-1 tensor, $a_{i,j,k} = c \cdot m_i x_j t_k$, where c is a constant depending on the chemical properties of the fluorophore. A mixture of r distinct pure fluorophores in varying concentrations will then admit an exact r-term CP decomposition. By computing a CP decomposition of the observed intensities in an unknown mixture of fluorophores, the time-varying emission-excitation spectra of the individual fluorophores can be separated, allowing a trained chemist to identify them.

In applications, a tensor is often corrupted by measurement and modeling errors, so that it does not admit an exact CP decomposition of small rank; however, often it may still be approximated well by such a decomposition. Therefore, the basic goal in practice consists of minimizing the objective function

$$f(V_1, V_2, \ldots, V_d) = \frac{1}{2} \left\| \mathcal{A} - \sum_{i=1}^{r} \mathbf{v}_1^{(i)} \otimes \cdots \otimes \mathbf{v}_d^{(i)} \right\|_F^2, \tag{1.1}$$

where $V_k = [\mathbf{v}_k^{(i)}]_{i=1}^r$ and $\| \cdot \|_F$ is the Frobenius norm, i.e., the square root of the sum of squares of elements of the tensor. Presently, two broad classes of iterative methods exist for tackling this problem: alternating least squares (ALS) methods, which alternately compute optimal V_k, such as the methods in [8, 20, 30, 31], and gradient-based optimization algorithms, such as the algorithms in [3, 4, 11, 21, 28, 29, 32, 36, 37]. ALS-type algorithms are generally efficient in terms of execution time, and they are easy to implement. The optimization-based methods, on the other hand, require a more involved computer implementation, but broad evidence suggests that they result in more accurate decompositions and faster convergence than ALS-type methods, particularly for difficult scenarios [3, 24, 36, 38]. From the complexity analysis in [36, Appendix A], it follows that the traditional computation of the gradient of the above objective function, which is required in every iteration of a

gradient-based optimization algorithm, contributes significantly to the computational complexity. Following [31], we will refer to this gradient as the CP gradient (CPG), and efficiently computing it is the topic of this paper.

If memory requirements are not a concern, Phan, Tichavský, and Cichocki [31] recently proposed an algorithm for efficiently computing the CPG, among other things. Their scheme exploits the substantial overlap that occurs in the computation of the individual components of the CPG, hereby essentially decreasing the computational cost from the naive $dr\prod_j n_j$ to $2r\prod_j n_j$. Their algorithm concurrently reduces the temporary memory requirements from the naive $\prod_j n_j + (r\prod_j n_j)/(\min_j n_j)$ to the much improved $\min_{1\le s\le d} \max\{\prod_{j\le s} n_j, \prod_{j>s} n_j\}$. In general, this reduces the temporary memory requirements significantly with respect to the standard implementation of the CPG; however, the cost may still be very high. Consider, for instance, cubic tensors with $n_1 = \cdots = n_d = n$; then the CPG is a set of d matrices of size $n \times r$, i.e., the space complexity is linear in d, r, and n, while the current state-of-the-art by [31] requires an amount of memory that is linear in r but at least *quadratic* in n and *exponential* in d, namely, $\mathcal{O}(rn^{\lceil d/2\rceil})$. Particularly for third-order tensors this cost may be excessively high, relative to the cost of storing the tensor, as $\mathcal{O}(rn^2)$ values need to be stored.

The main contribution of our undertaking is a time and memory efficient implementation of the CPG based on tensor blocking techniques. The proposed algorithm reduces the memory usage to a moderate constant.[1] The algorithm is shown to attain upward of 90% of the peak performance of the computer system employed in our tests, while concurrently improving the memory consumption with respect to the unblocked algorithm of [31] by up to two orders of magnitude for large dense tensors.

The paper is structured as follows. In the next section, we recall some results from the literature. Section 3 summarizes the approach for computing the CPG from [31], which is the basic algorithm that will be applied at the block level in the proposed algorithm. In section 4, we show that the CPG can be computed blockwise without data reorganization and propose a heuristic for choosing the block sizes automatically. Some important implementation details are considered in section 5. The proposed blockwise algorithm and the algorithm of [31] are experimentally compared in section 6. Finally, section 7 presents our conclusions.

2. Preliminaries. We recall some basic properties about tensors, unfoldings, and tensor-to-vector contractions from the literature.

Concerning notation, the following conventions apply throughout the manuscript. Vectors are typeset in a bold face font ($\mathbf{v}$), matrices in upper case (M, V), and tensors in a calligraphic font ($\mathcal{A}$, $\mathcal{B}$). The order of a tensor is always denoted by d. The identity matrix of order n is denoted by I_n; the subscript may be dropped if it is clear from the context. We define the columnwise Khatri–Rao product as $A \odot B := \begin{bmatrix}\mathbf{b}_1 \otimes_K \mathbf{a}_1 & \cdots & \mathbf{b}_n \otimes_K \mathbf{a}_n\end{bmatrix}$, where $\mathbf{a}_i$ and $\mathbf{b}_i$ are the columns of A and B, respectively, and where $\otimes_K$ is the usual Kronecker product $(\mathbf{b}\otimes_K\mathbf{a})^T := \begin{bmatrix}b_1\mathbf{a}^T & \cdots & b_m\mathbf{a}^T\end{bmatrix}$.

2.1. Tensors. A d-array $\mathcal{A} = [a_{i_1,\ldots,i_d}]_{i_1,\ldots,i_d=1}^{n_1,\ldots,n_d} \in \mathbb{R}^{n_1\times\cdots\times n_d}$ can be considered as a coordinate representation of an abstract tensor $\mathcal{A}$ with respect to the standard tensor basis $\{\mathbf{e}^1_{i_1} \otimes \mathbf{e}^2_{i_2} \otimes \cdots \otimes \mathbf{e}^d_{i_d}\}_{i_1,i_2,\ldots,i_d=1}^{n_1,n_2,\ldots,n_d}$; herein, $\mathbf{e}^k_{i_k}$ is the i_kth standard basis

[1] The standard approach in the literature is to define the memory complexity of an algorithm as the maximum number of memory used by the algorithm, excluding the cost for storing the input and output of the algorithm.

vector of $\mathbb{R}^{n_k}$. In this manner, $\mathcal{A}$ may formally be written as

$$\mathcal{A} = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \cdots \sum_{i_d=1}^{n_d} a_{i_1,i_2,\ldots,i_d} \mathbf{e}_{i_1}^1 \otimes \mathbf{e}_{i_2}^2 \otimes \cdots \otimes \mathbf{e}_{i_d}^d. \tag{2.1}$$

We make no notational distinction between the abstract tensor $\mathcal{A}$ that lives in $\mathbb{R}^{n_1} \otimes \cdots \otimes \mathbb{R}^{n_d}$, i.e., the tensor product of vector spaces, and its coordinate representation, the array $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$. We will refer to $\mathbb{R}^{n_i}$ as the ith *factor* of this tensor product of vector spaces and to n_i as the size of the ith factor.

A multilinear transformation from $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ to $\mathcal{B} \in \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_d}$ via a set of matrices $\{M_k \in \mathbb{R}^{m_k \times n_k}\}_{k=1}^d$ is defined as

$$\mathcal{B} := \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \cdots \sum_{i_d=1}^{n_d} a_{i_1,i_2,\ldots,i_d} (M_1 \mathbf{e}_{i_1}^1) \otimes (M_2 \mathbf{e}_{i_2}^2) \otimes \cdots \otimes (M_d \mathbf{e}_{i_d}^d).$$

Following [10], we write this operation as

$$\mathcal{B} = (M_1, M_2, \ldots, M_d) \cdot \mathcal{A} = (M_1^T, M_2^T, \ldots, M_d^T)^T \cdot \mathcal{A}.$$

It is linear in every factor.

2.2. Unfoldings. *Unfoldings*, *matricizations*, or *flattenings* are a central idea in tensor decompositions. From a practical perspective, the motivation is transforming tensor operations into familiar operations on matrices so as to take advantage of optimized libraries implementing the BLAS interface [13, 14].

The *explicit unfolding* of a tensor $\mathcal{A}$ in factor k, which is denoted by $\mathcal{A}_{(k)}$, results in an $n_k \times \prod_{i \neq k} n_i$ matrix whose columns are the mode-k vectors of $\mathcal{A}$; a mode-k vector $\mathbf{v}$ is a vector that is obtained by fixing all indices of $\mathcal{A}$ while varying only the index in factor k, i.e., $\mathbf{v} = \mathcal{A}_{i_1,\ldots,i_{k-1},:,i_{k+1},\ldots,i_d}$ with i_j a fixed value. The ordination of the mode-k vectors in the unfolding is determined by definition; we assume the canonical unfolding from [15] in this paper. Additionally, we assume that a tensor $\mathcal{A}$ is stored as the vectorization of the mode-1 unfolding, which is called the *canonical vectorization* and is denoted by $\operatorname{vec}(\mathcal{A})$. Mode-$k$ unfoldings generally require an explicit reorganization of the data elements of the tensor, hereby necessitating the allocation of additional memory for storing the unfolding if one is to employ the aforementioned optimized matrix libraries. Additionally, the memory access pattern is neither linear nor (exclusively) strided; hence several index calculations are necessary, thus impeding expeditious execution; an indication of the expected performance loss is presented in Tables 1, 2, and 3 in [39].

Implicit unfoldings, on the other hand, do not require a permutation of the data elements of the tensor for obtaining the unfolding. Recall from [15, section 2.2] that

$$\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)} \in \mathbb{R}^{n_1 \cdots n_k \times n_{k+1} \cdots n_d} \quad \text{is defined by} \quad (\mathcal{A}_{(1,\ldots,k;k+1,\ldots,d)})_{\imath,\jmath} := a_{i_1,\ldots,i_d},$$

where

$$\imath = 1 + \sum_{l=1}^{k} (i_{k-l+1} - 1) \prod_{l'=1}^{l-1} n_{k-l'+1} \quad \text{and} \quad \jmath = 1 + \sum_{l=1}^{d-k} (i_{d-l+1} - 1) \prod_{l'=1}^{l-1} n_{d-l'+1};$$

we assume that an empty sum equals 0 and an empty product equals 1, so that the above is well defined for all $k = 0, \ldots, d$. With this unfolding, the consecutive set of

factors $1,\dots,k$ is mapped to the rows of the unfolded matrix and $k+1,\dots,d$ to the columns. From the above definition, it can be verified, using straightforward computations, that these unfoldings have the interesting property that $\mathcal{A}_{(1,\dots,k;k+1,\dots,d)} = \mathcal{A}^T_{(k+1,\dots,d;1,\dots,k)}$, and, furthermore,

$$\operatorname{vec}(\mathcal{A}) := \mathcal{A}_{(1,\dots,d;\varnothing)} = \operatorname{vec}\left(\mathcal{A}_{(1,\dots,k;k+1,\dots,d)}\right), \quad k = 0,\dots,d.$$

This last equation entails that $\operatorname{vec}(\mathcal{A})$ can be interpreted as the column-major linearization of $\mathcal{A}_{(1,\dots,k;k+1,\dots,d)}$ for $k = 0,\dots,d$. In this case, neither explicit reorganization of the data elements of the tensor nor index calculations are required. Note that the only mode-k unfoldings that are also implicit are $k = 1, d-1$.

As an illustration of these two types, consider the tensor $\mathcal{A} \in \mathbb{R}^{4\times3\times2}$, where $a_{ijk} = 12(k-1) + 4(j-1) + i$. Its canonical vectorization is

$$\operatorname{vec}(\mathcal{A}) = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \end{bmatrix}^T,$$

and its unfoldings are given by

$$\mathcal{A}_{(1)} = \mathcal{A}_{(1;2,3)} = \begin{bmatrix} 1 & 5 & 9 & 13 & 17 & 21 \\ 2 & 6 & 10 & 14 & 18 & 22 \\ 3 & 7 & 11 & 15 & 19 & 23 \\ 4 & 8 & 12 & 16 & 20 & 24 \end{bmatrix}, \mathcal{A}_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 13 & 14 & 15 & 16 \\ 5 & 6 & 7 & 8 & 17 & 18 & 19 & 20 \\ 9 & 10 & 11 & 12 & 21 & 22 & 23 & 24 \end{bmatrix}, \text{ and}$$

$$\mathcal{A}_{(3)} = \mathcal{A}^T_{(1,2;3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \end{bmatrix};$$

the remaining implicit unfoldings are $\mathcal{A}^T_{(\varnothing;1,2,3)} = \operatorname{vec}(\mathcal{A}) = \mathcal{A}_{(1,2,3;\varnothing)}$. Note that the column-major linearizations of the implicit unfoldings indeed coincide with $\operatorname{vec}(\mathcal{A})$. The mode-2 unfolding cannot be obtained without data permutations, hence requiring some additional memory for storing the unfolding if one wishes to compute (3.2).

2.3. Tensor-to-vector contractions. The *factor-k tensor-to-vector contraction* (k-TVC) is a special type of multilinear transformation which is defined as

$$\mathbf{w}_k = (\mathbf{v}_1,\dots,\mathbf{v}_{k-1},I_{n_k},\mathbf{v}_{k+1},\dots,\mathbf{v}_d)^T \cdot \mathcal{A}, \quad \text{where } \mathbf{v}_i \in \mathbb{R}^{n_i} \text{ and } \mathbf{w}_k \in \mathbb{R}^{n_k}.$$

It can be computed with successive matrix-vector products using a technique from [31] that we will refer to as *successive contractions*. The idea is to write

$$\mathbf{w}_k = (I,\dots,I,\mathbf{v}_{k+1},\dots,\mathbf{v}_d)^T \cdot \Big(\underbrace{(\mathbf{v}_1,\dots,\mathbf{v}_{k-1},I,\dots,I)^T \cdot \mathcal{A}}_{\mathcal{B}}\Big).$$

Let $\mathcal{B}^0 = \mathcal{A}$. Then, the *left-to-right contraction* yielding $\mathcal{B}$ may be computed by

$$\mathcal{B}^\ell_{(1,\dots,\ell;\ell+1,\dots,d)} \leftarrow \mathbf{v}_\ell^T \mathcal{B}^{\ell-1}_{(1,\dots,\ell;\ell+1,\dots,d)}, \quad \ell = 1,\dots,k-1,$$

where we interpret $\mathcal{B}^\ell \in \mathbb{R}^{1\times\cdots\times1\times n_{\ell+1}\times\cdots\times n_d}$, resulting in $\mathcal{B} = \mathcal{B}^{k-1}$. This scheme is then followed by a *right-to-left contraction*, which computes

$$\mathcal{C}^\ell_{(1,\dots,d-\ell;d-\ell+1,\dots,d)} \leftarrow \mathcal{C}^{\ell-1}_{(1,\dots,d-\ell;d-\ell+1,\dots,d)} \mathbf{v}_{d-\ell+1}, \quad \ell = 1,\dots,d-k,$$

with $\mathcal{C}^\ell \in \mathbb{R}^{1\times\cdots\times1\times n_k\times\cdots\times n_{d-\ell}\times1\times\cdots\times1}$ starting from $\mathcal{C}^0 = \mathcal{B}$. Then, the result $\mathbf{w}_k = \mathcal{C}^{d-k} \in \mathbb{R}^{1\times\cdots\times1\times n_k\times1\times\cdots\times1} \cong \mathbb{R}^{n_k}$.

For future reference, we summarize a procedure for computing the right-to-left contraction in Algorithm 1. Note that only implicit unfoldings are required in this procedure. In particular, the computation in step 3 can be accomplished practically by providing the correct stride length, i.e., $n_1 \cdots n_{j-1}$, to the BLAS2 routine `dgemv`. Also note that after the multiplication with $\mathbf{v}_j$, we interpret the resulting vector as a tensor of order $j-1$; by the time the algorithm reaches step 5, the tensor $\mathcal{B}$ has been reduced to a vector of length n_1.

ALGORITHM 1: Computing a complete right-to-left contraction (RTLC).

input : $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and $\{\mathbf{v}_j \in \mathbb{R}^{n_j}\}_{j=2}^{d}$
output: $\mathbf{w} = (I, \mathbf{v}_2, \ldots, \mathbf{v}_d)^T \cdot \mathcal{A} \in \mathbb{R}^{n_1}$
1 $\mathcal{B}_{(1,\ldots,d-1;d)} \leftarrow \mathcal{A}_{(1,\ldots,d-1;d)}\mathbf{v}_d$
2 **for** $j = d-1, d-2, \ldots, 2$ **do**
3 $\quad \mathcal{B}_{(1,\ldots,j-1;j)} \leftarrow \mathcal{B}_{(1,\ldots,j-1;j)}\mathbf{v}_j$
4 **end**
5 $\mathbf{w} \leftarrow \mathcal{B}$

3. CP gradient. We define a set of *multiple k-TVCs* (k-MTVC) as

$$\mathbf{w}_k^{(i)} = (\mathbf{v}_1^{(i)}, \ldots, \mathbf{v}_{k-1}^{(i)}, I, \mathbf{v}_{k+1}^{(i)}, \ldots, \mathbf{v}_d^{(i)})^T \cdot \mathcal{A}, \quad i = 1, \ldots, r, \tag{3.1}$$

where $\mathbf{w}_k^{(i)} \in \mathbb{R}^{n_k}$, and where the vectors $\{\mathbf{v}_j^{(i)} \in \mathbb{R}^{n_j}\}_{i=1}^{r}$, $j = 1, \ldots, d$, are all simultaneously available. If we define for $j = 1, \ldots, d$ the matrices

$$V_j = \begin{bmatrix} \mathbf{v}_j^{(1)} & \cdots & \mathbf{v}_j^{(r)} \end{bmatrix} \in \mathbb{R}^{n_j \times r} \text{ and } W_j = \begin{bmatrix} \mathbf{w}_j^{(1)} & \cdots & \mathbf{w}_j^{(r)} \end{bmatrix} \in \mathbb{R}^{n_j \times r},$$

then it follows that the k-MTVC (3.1) can be computed equivalently as

$$W_k \leftarrow \mathcal{A}_{(k)}(V_1 \odot \cdots \odot V_{k-1} \odot V_{k+1} \odot \cdots \odot V_d). \tag{3.2}$$

This particular product is often called a matricized-tensor times Khatri–Rao product (MTTKRP) in the literature [25]. MTVCs appear in ALS algorithms for constructing a CP decomposition as in (1.1); it is typically the operation with the dominant cost in the algorithms of [8, 20, 29, 31]. The CPG can be defined in terms of k-MTVCs: it is well understood that the CPG of the objective function (1.1) is given by the set of matrices $\{W_k\}_{k=1}^{d}$. The straightforward implementation of the CPG computes the matrices W_k as is suggested by (3.2): first compute the Khatri–Rao products, resulting in a huge matrix, then *compute* the mode-k unfolding of $\mathcal{A}$, resulting in another huge matrix, and finally use the BLAS3 `dgemm` routine for computing the matrix product.

Phan, Tichavský, and Cichocki [31] recently presented an efficient technique based on implicit unfoldings for computing k-MTVCs. Their key observation is that, for a fixed *splitting point* $1 \leq s \leq d$, the tensors

$$\mathcal{B}^{(i)} = (\mathbf{v}_1^{(i)}, \ldots, \mathbf{v}_s^{(i)}, I, \ldots, I)^T \cdot \mathcal{A} \quad \text{and} \quad \mathcal{C}^{(i)} = (I, \ldots, I, \mathbf{v}_{s+1}^{(i)}, \ldots, \mathbf{v}_d^{(i)})^T \cdot \mathcal{A},$$

for $i = 1, \ldots, r$, can be computed efficiently by, respectively,

$$\mathbb{R}^{n_{s+1} \cdots n_d \times r} \ni B \leftarrow \mathcal{A}^T_{(1,\ldots,s;s+1,\ldots,d)}(V_1 \odot \cdots \odot V_s) \quad \text{and} \tag{3.3a}$$

$$\mathbb{R}^{n_1 \cdots n_s \times r} \ni C \leftarrow \mathcal{A}_{(1,\ldots,s;s+1,\ldots,d)}(V_{s+1} \odot \cdots \odot V_d); \tag{3.3b}$$

the ith columns of B and C then correspond with $\operatorname{vec}\left(\mathcal{B}^{(i)}\right)$ and $\operatorname{vec}\left(\mathcal{C}^{(i)}\right)$, respectively. The kth component of the CPG, i.e., the k-MTVC, $k > s$, is then efficiently constructed by computing B, followed by applying successive contractions for computing the k-TVCs with the order-$(d-s)$ tensors $\mathcal{B}^{(i)}$, $i = 1, \ldots, r$.[2] An analogous result holds for k-MTVCs with $k \leq s$, replacing B by C and $\mathcal{B}^{(i)}$ by $\mathcal{C}^{(i)}$. This scheme for computing an individual component W_k of the CPG involves only implicit unfoldings, hereby eliminating the need for storing the unfolding as is required by the naive implementation based on (3.2).

For computing all components of the CPG, [31] suggests exploiting the overlap that exists between the individual k-MTVCs, $k = 1, \ldots, d$. Algorithm 2 in [31] essentially operates as follows. First, compute the k-MTVCs, $k = s+1, \ldots, d$, by constructing B once, and then performing the remaining k-TVCs with $\mathcal{B}^{(i)}$, $i = 1, \ldots, r$, by applying a left-to-right contraction followed by a right-to-left contraction as explained in section 2.3. Then, in computing these k-TVCs some additional operations can be eliminated in the left-to-right contraction, namely by temporarily storing $(\mathcal{B}^{(i)})^{\ell-1}$, $i = 1, \ldots, r$, when computing the ℓ-TVCs. In the next set of TVCs, i.e., the $(\ell+1)$-TVCs, the left-to-right contraction can then simply proceed from these temporarily stored tensors. The k-MTVCs with $k = 1, \ldots, s$ are computed by forming C once, and then performing the remaining k-TVCs with $\mathcal{C}^{(i)}$, $i = 1, \ldots, r$, again exploiting the observation that some operations can be spared in the left-to-right contractions.

For future reference, the above-mentioned version of [31, Algorithm 2] is formalized in Algorithm 2. The call to RTLC in lines 8 and 18 invokes Algorithm 1. Only implicit unfoldings are employed in this routine. The matrix products and matrix-vector products featured in the algorithm may thus be computed by providing the correct stride length to the BLAS routines `dgemm` and `dgemv`, respectively.

Time and space complexity. We will assume that the Khatri–Rao product $V_1 \odot \cdots \odot V_s$ is evaluated from left to right as $((\cdots((V_1 \odot V_2) \odot V_3) \odot \cdots) \odot V_s)$ and similarly for the Khatri–Rao product in line 11 of Algorithm 2. It is assumed that an algorithm is employed that computes the Khatri–Rao product $A \odot B$ of $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{n \times r}$ in precisely mnr operations; such an algorithm is considered in section 5. Assume, without loss of generality, that $n_1 \cdots n_s > n_{s+1} \cdots n_d$, and then the number of floating-point operations of Algorithm 3 can be bounded from above by

$$\underbrace{4rn_1 \cdots n_d}_{\text{2 matrix products}} + \underbrace{r(d-2)n_1 \cdots n_s}_{\text{2 Khatri–Rao products}} + \underbrace{2r(d-2)n_1 \cdots n_s}_{\text{2}r\text{ successive contractions}};$$

herein, the cost of the Khatri–Rao products in lines 1 and 11 was bounded individually as follows:

$$n_1 n_2 r + n_1 n_2 n_3 r + \cdots + n_1 n_2 \cdots n_s r < r(s-1) n_1 \cdots n_s, \text{ and} \tag{3.4a}$$

$$n_{s+1} n_{s+2} r + n_{s+1} n_{s+2} n_{s+3} r + \cdots + n_{s+1} \cdots n_d r < r(d-s-1) n_{s+1} \cdots n_d. \tag{3.4b}$$

It can be verified that the total cost of the successive contractions is twice the amounts on the left-hand sides of (3.4). From these bounds it follows that the cost of the Khatri–Rao products and successive contractions may be overstated asymptotically; if $n_1, \ldots, n_d \to \infty$ with $c_1 < n_j/n_i < c_2$, $1 \leq i, j \leq d$, for some constants $c_1 > 0$ and $c_2 < \infty$, then we can discard the lower-order terms on the left-hand side of (3.4), so

[2] The scheme discussed here differs slightly from the original presentation in [31], but the asymptotic time complexity is unaltered.

ALGORITHM 2: Computing the CPG following [31] (PTC-CPG).

input : $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and $\{V_j \in \mathbb{R}^{n_j \times r}\}_{j=1}^{d}$
output: $\{W_j = \mathcal{A}_{(j)}(V_1 \odot \cdots \odot V_{j-1} \odot V_{j+1} \odot \cdots \odot V_d) \in \mathbb{R}^{n_j \times r}\}_{j=1}^{d}$
$B \leftarrow \mathcal{A}^T_{(1,\ldots,s;s+1,\ldots,d)}(V_1 \odot V_2 \odot \cdots \odot V_s)$
$\mathcal{B}^{(i,s+1)} \leftarrow \mathbf{b}_i, \quad i = 1, \ldots, r,$
for $k = s+1, \ldots, d$ **do**
  for $i = 1, \ldots, r$ **do**
    if $k > s+1$ **then**
      $\mathcal{B}^{(i,k)}_{(1,\ldots,k-1;k,\ldots,d)} \leftarrow (\mathbf{v}^{(i)}_{k-1})^T \mathcal{B}^{(i,k-1)}_{(1,\ldots,k-1;k,\ldots,d)}$
    end
    $\mathbf{w}^{(i)}_k \leftarrow \mathrm{RTLC}(\mathcal{B}^{(i,k)}, \{\mathbf{v}^{(i)}_{k+1}, \ldots, \mathbf{v}^{(i)}_d\})$
  end
end
$C \leftarrow \mathcal{A}_{(1,\ldots,s;s+1,\ldots,d)}(V_{s+1} \odot V_{s+2} \odot \cdots \odot V_d)$
$\mathcal{C}^{(i,1)} \leftarrow \mathbf{c}_i, \quad i = 1, \ldots, r,$
for $k = 1, \ldots, s$ **do**
  for $i = 1, \ldots, r$ **do**
    if $k > 1$ **then**
      $\mathcal{C}^{(i,k)}_{(1,\ldots,k-1;k,\ldots,d)} \leftarrow (\mathbf{v}^{(i)}_{k-1})^T \mathcal{C}^{(i,k-1)}_{(1,\ldots,k-1;k,\ldots,d)}$
    end
    $\mathbf{w}^{(i)}_k \leftarrow \mathrm{RTLC}(\mathcal{C}^{(i,k)}, \{\mathbf{v}^{(i)}_{k+1}, \ldots, \mathbf{v}^{(i)}_s\})$
  end
end

that we obtain the asymptotic time complexity

$$\mathcal{O}\Big(4r\prod_{j=1}^{d} n_j + 6r \cdot \max\{\prod_{j=1}^{s} n_j, \prod_{j=s+1}^{d} n_j\}\Big) = \mathcal{O}\Big(4r\big(1 + \tfrac{3}{2}\max\{\prod_{j=1}^{s} \tfrac{1}{n_j}, \prod_{j=s+1}^{d} \tfrac{1}{n_j}\}\big)\prod_{j=1}^{d} n_j\Big).$$

From this formula, one learns that it is beneficial to choose a splitting point $1 \leq s \leq d$ such that $n_1 \cdots n_s \approx n_{s+1} \cdots n_d$ for minimizing the number of operations.[3]

The memory requirements of the algorithm, i.e., those requirements in excess of storing the input and output, are asymptotically of the order

$$\mathcal{O}\Big(r\prod_{i=1}^{s} n_j + r\prod_{i=s+1}^{d} n_j\Big),$$

which includes the cost for storing the Khatri–Rao structured matrix and the result of the matrix multiplication, i.e., B or C. The aforementioned heuristic for choosing the splitting point is equally beneficial for minimizing the memory cost.

4. Blocked algorithm. We consider blocking strategies for the k-MTVC to the end of designing an algorithm for computing the CPG that consumes only a constant

[3]Note that the naive approach of computing the CPG via (3.2) has a time complexity that grows asymptotically as $2dr\prod_{j=1}^{d} n_j$.

amount of memory. Blocking for matrices is well understood; however, for tensors it was considered only quite recently. In 2011, Phan and Cichocki [30] considered an explicit division of a tensor into blocks for computing a CP decomposition using a hierarchical ALS-type algorithm that is applicable to dense large-scale tensors. A theoretical contribution was made in 2012 by Ragnarsson and Van Loan [33], who were chiefly concerned with the connection between blocked tensors and an unfolding resulting in a matrix with a block structure. One year later, they extended their results in [34] by proposing a technique for embedding an unsymmetric tensor into a symmetric tensor of larger size. Schatz et al. [35] proposed a blocking strategy for efficiently storing symmetric tensors and applying a symmetric multilinear multiplication in 2014. The idea of subdividing the CP approximation problem into smaller independent problems was also considered by Hansen, Plantenga, and Kolda [19] in the context of large-scale sparse tensors. Software libraries, which were developed for quantum chemistry applications, for working with tensors that also support divisions into subtensors include libtensor [16], the Tensor Contraction Engine [6], and TiledArrays [2].

First, it is shown that both the k-MTVC and CPG may be computed blockwise in constant memory—however, at a slightly increased computational complexity. Then, in section 4.2, a particular choice of division into subtensors is proposed, such that one particular implicit unfolding corresponds with the block unfolding of [33]. Algorithmically choosing the size of the subtensors so that the CPG is computed within a user-specified memory consumption while limiting the time complexity is addressed in section 4.3.

4.1. Blockwise computation. The basic result we exploit is that the k-MTVC of a blocked tensor may be computed through k-MTVCs with each of the subtensors. Based on this result, we may similarly construct the CPG of a blocked tensor by computing the CPGs at the block level and aggregating the local results.

Let $\mathcal{A}$ be as in (2.1), and assume that we subdivide it into $q_1 \times \cdots \times q_d$ blocks of size $b_1 \times \cdots \times b_d$: we formally write

$$\mathcal{A} = \left[\mathcal{A}^{i_1,i_2,\ldots,i_d}\right]_{i_1,i_2,\ldots,i_d=1}^{q_1,q_2,\ldots,q_d}, \quad \text{where} \quad \mathcal{A}^{i_1,i_2,\ldots,i_d} \in \mathbb{R}^{b_1 \times b_2 \times \cdots \times b_d}, \tag{4.1}$$

i.e., $b_j q_j = n_j$, which we shall assume for the sake of simplicity.[4] Consider a matrix $S_{j_k}^T \in \mathbb{R}^{b_k \times n_k}$, $j_k = 1, \ldots, q_k$, that "selects" the rows $(j_k - 1)b_k + 1$ through $j_k b_k$ when applied to a matrix with compatible dimensions, i.e.,

$$S_{j_k}^T = \begin{bmatrix} O_1 & \cdots & O_{j_k-1} & I_{b_k} & O_{j_k+1} & \cdots & O_{q_k} \end{bmatrix}, \tag{4.2}$$

where I_{b_k} is the $b_k \times b_k$ identity matrix and every O_i is the $b_k \times b_k$ zero matrix. Using these definitions, one finds that

$$\begin{aligned}(S_{j_1}, \ldots, S_{j_d})^T \cdot \mathcal{A} &:= (S_{j_1}, \ldots, S_{j_d})^T \cdot \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} (\mathbf{e}_{i_1}, \ldots, \mathbf{e}_{i_d}) \cdot a_{i_1,\ldots,i_d} \\ &= \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} (S_{j_1}^T \mathbf{e}_{i_1}, \ldots, S_{j_d}^T \mathbf{e}_{i_d}) \cdot a_{i_1,\ldots,i_d}\end{aligned}$$

[4] Extensions to accommodate for fringe blocks and nonuniform blocking patterns are left as an exercise; the code we developed implements the former but not the latter extension.

$$= \sum_{i_1=(j_1-1)b_1+1}^{j_1 b_1} \cdots \sum_{i_d=(j_d-1)b_d+1}^{j_d b_d} (\mathbf{e}'_{i_1}, \ldots, \mathbf{e}'_{i_d}) \cdot a_{i_1,\ldots,i_d} = \mathcal{A}^{j_1,\ldots,j_d},$$

where $\mathbf{e}'_{i_k}$ is the $(i_k - (j_k - 1)b_k)$th standard basis vector in $\mathbb{R}^{b_k}$. Note that we dropped the superscript in $\mathbf{e}_{i_k}$ relative to (2.1) for the sake of brevity. In the above equations, the second equality is due to the multilinearity of the product, and the penultimate equality follows from straightforward computations. Consider, for the sake of notational brevity but without loss of generality, the 1-MTVCs, $i = 1, \ldots, r$,

$$\begin{aligned}
\mathbf{w}_1^{(i)} &= (I, \mathbf{v}_2^{(i)}, \mathbf{v}_3^{(i)}, \ldots, \mathbf{v}_d^{(i)})^T \cdot \mathcal{A} \\
&= (I, \mathbf{v}_2^{(i)}, \mathbf{v}_3^{(i)}, \ldots, \mathbf{v}_d^{(i)})^T \cdot \Big(I, \sum_{j_2=1}^{q_2} S_{j_2} S_{j_2}^T, \ldots, \sum_{j_d=1}^{q_d} S_{j_d} S_{j_d}^T\Big) \cdot \mathcal{A} \\
&= \sum_{j_2=1}^{q_2} \cdots \sum_{j_d=1}^{q_d} (I, S_{j_2}^T \mathbf{v}_2^{(i)}, \ldots, S_{j_d}^T \mathbf{v}_d^{(i)})^T \cdot (I, S_{j_2}, \ldots, S_{j_d})^T \cdot \mathcal{A}, \qquad (4.3)
\end{aligned}$$

where in the second equality we note that the orthogonal projectors $S_{j_k} S_{j_k}^T$ sum to the identity matrix. In the last equality the multilinearity property was exploited several times to move the sums out of the multiplication. Partition $\mathbf{v}_k^{(i)}$ as

$$(\mathbf{v}_k^{(i)})^T = \Big[(\mathbf{v}_{k,1}^{(i)})^T \ \cdots \ (\mathbf{v}_{k,q_k}^{(i)})^T\Big], \quad \text{where} \quad \mathbf{v}_{k,j}^{(i)} \in \mathbb{R}^{b_k},\ j = 1, \ldots, q_k, i = 1, \ldots, r,$$

and partition $\mathbf{w}_k^{(i)}$ analogously. Then, we apply $S_{j_1}^T$ on both sides of (4.3) to obtain

$$\begin{aligned}
\mathbf{w}_{1,j_1}^{(i)} &= \sum_{j_2=1}^{q_2} \cdots \sum_{j_d=1}^{q_d} (I, \mathbf{v}_{2,j_2}^{(i)}, \ldots, \mathbf{v}_{d,j_d}^{(i)})^T \cdot (S_{j_1}, S_{j_2}, \ldots, S_{j_d})^T \cdot \mathcal{A} \\
&= \sum_{j_2=1}^{q_2} \cdots \sum_{j_d=1}^{q_d} (I, \mathbf{v}_{2,j_2}^{(i)}, \ldots, \mathbf{v}_{d,j_d}^{(i)})^T \cdot \mathcal{A}^{j_1,j_2,\ldots,j_d}.
\end{aligned}$$

In general, having subdivided $\mathcal{A}$ as in (4.1), the k-MTVC in (3.1) may thus be computed blockwise by computing the $\prod_{j=1}^d q_j$ k-MTVCs

$$(\mathbf{v}_{1,j_1}^{(i)}, \ldots, \mathbf{v}_{k-1,j_{k-1}}^{(i)}, I, \mathbf{v}_{k+1,j_{k+1}}^{(i)}, \ldots, \mathbf{v}_{d,j_d}^{(i)})^T \cdot \mathcal{A}^{j_1,\ldots,j_d}, \quad i = 1, \ldots, r, \qquad (4.4)$$

$j_l = 1, \ldots, q_l$ with $l = 1, \ldots, d$, and summing all of the obtained vectors, for a fixed j_k, to form $\mathbf{w}_{k,j_k}^{(i)}$. An alternative derivation of this result can be obtained by deriving the objective function (1.1) to the submatrices of V_k, as was done in [30, section 3].

For computing the k-MTVCs at the block level, we use the algorithm from [31], which was described in section 3 and presented as Algorithm 2, with one minor modification: for ensuring that the memory consumption of the algorithm is constant, the k-MTVC in (4.4) should be subdivided into a sequence of γ k-MTVCs, each of which involves an approximately equal and constant number of vectors. That is, compute (4.4) as

$$(\mathbf{v}_{1,j_1}^{(i)}, \ldots, \mathbf{v}_{k-1,j_{k-1}}^{(i)}, I, \mathbf{v}_{k+1,j_{k+1}}^{(i)}, \ldots, \mathbf{v}_{d,j_d}^{(i)})^T \cdot \mathcal{A}^{j_1,\ldots,j_d}, \quad i = t_l + 1, \ldots, t_{l+1}, \qquad (4.5)$$

where $0 = t_0 < t_1 < \cdots < t_\gamma = r$. Letting r_0 be a constant chosen by the user, it is clear that we can always choose a sequence as above such that $(t_l - t_{l-1}) \leq r_0$ for

ALGORITHM 3: Blockwise computation of the CPG in constant memory.

input : $\mathcal{A}$ as in (4.1) and $\{V_j \in \mathbb{R}^{n_j \times r}\}_{j=1}^{d}$
output: $\{W_j = \mathcal{A}_{(j)}(V_1 \odot \cdots \odot V_{j-1} \odot V_{j+1} \odot \cdots \odot V_d) \in \mathbb{R}^{n_j \times r}\}_{j=1}^{d}$
1 $W_j \leftarrow 0,\ j = 1, 2, \ldots, d,$
2 **for** $(j_1, j_2, \ldots, j_d) = (1, 1, \ldots, 1), \ldots, (q_1, q_2, \ldots, q_d)$ **do**
3 **for** $i = 1, \ldots, \gamma$ **do**
4 $\{Z_k\}_{k=1}^{d} \leftarrow$ PTC-CPG($\mathcal{A}^{j_1, j_2, \ldots, j_d}, \{S_{j_k}^T V_k \widehat{S}_i\}_{k=1}^{d}$)
5 **for** $k = 1, \ldots, d$ **do**
6 $W_k \leftarrow W_k + S_{j_k} Z_k \widehat{S}_i^T$
7 **end**
8 **end**
9 **end**

every $l = 1, \ldots, \gamma$. *The process of subdividing one k-MTVC into a sequence of several k-MTVCs with fewer vectors will be referred to as sequencing.*

For realizing a blockwise computation of the CPG, it suffices to understand that for a fixed choice of $j_1, j_2, \ldots, j_d$, the k-MTVCs, $k = 1, \ldots, d$, in (4.4) and (4.5) are all independent. We can compute the CPG of $\mathcal{A}^{j_1, j_2, \ldots, j_d}$ and the matrices $\{S_{j_l}^T V_l\}_{l=1}^{d}$ at the block level, obtaining a set of matrices $\{Z_l \in \mathbb{R}^{b_l \times r}\}_{l=1}^{d}$, all of which provide a partial contribution to the corresponding rows of the matrices in $\{W_l\}_{l=1}^{d}$. If the number of vectors $r > r_0$, then we apply sequencing to the block-level CPG so that we successively compute $\gamma \geq 2$ block-level CPGs. Let $\widehat{S}_i \in \mathbb{R}^{r \times (t_i - t_{i-1})}$, $i = 1, \ldots, \gamma$, be defined as

$$\widehat{S}_i = \begin{bmatrix} O_1 & \cdots & O_{i-1} & I_{t_i - t_{i-1}} & O_{i+1} & \cdots & O_\gamma \end{bmatrix}^T,$$

with $O_k = 0 \in \mathbb{R}^{(t_k - t_{k-1}) \times (t_k - t_{k-1})}$, $k = 1, \ldots, \gamma$, a matrix of zeros. Then, the proposed algorithm for computing the CPG blockwise in constant memory is formalized as Algorithm 3. Note that line 2 is executed $q_1 q_2 \cdots q_d$ times. In line 4 the call to PTC-CPG invokes Algorithm 2.

Time and space complexity. The asymptotic time complexity of the blockwise implementation is simply $\prod_{j=1}^{d} q_j$ times the time complexity of a CPG computed by Algorithm 2 on tensors of size (at most) $b_1 \times \cdots \times b_d$. Thus, we obtain

$$\mathcal{O}\Bigg(4r\big(1 + \tfrac{3}{2}\max\{\prod_{j=1}^{s} \tfrac{1}{b_j}, \prod_{j=s+1}^{d} \tfrac{1}{b_j}\}\big) \prod_{j=1}^{d} n_j\Bigg).$$

From this formula it is clear that the block sizes should be chosen so that $b_1 \cdots b_s \approx b_{s+1} \cdots b_d$ in order to minimize the number of operations with the blocked implementation. Comparing the above with the complexity of computing the CPG using Algorithm 2 from [31], we see that the number of operations in the blocked implementation will always be larger; however, if $b_1 \cdots b_s \approx b_{s+1} \cdots b_d$ is sufficiently large, then the relative increase in the total number of operations will be small.

The additional memory requirements are, indeed, constant:

$$\mathcal{O}\Bigg(r_0 \prod_{j=1}^{s} b_j + r_0 \prod_{j=s+1}^{d} b_j\Bigg).$$

4.2. Implicit block unfoldings. The algorithm of [31] for computing the CPG using implicit unfoldings requires that the elements of the subtensors $\mathcal{A}^{j_1,\ldots,j_d}$ appear in consecutive memory; however, with the canonical vectorization of $\mathcal{A}$ this is, in general, not the case.[5] We propose resolving this issue by considering a particular division into subtensors, so that there exists one particular implicit unfolding of $\mathcal{A}$ that equals its block unfolding. In this manner, the required implicit unfoldings with each of the subtensors can be realized as submatrices of the corresponding implicit unfolding of $\mathcal{A}$. An optimized matrix multiplication routine can then immediately be called for computing the product with the Khatri–Rao product matrix, by providing the correct stride length.

Let $\mathcal{A}$ be as in (4.1). Then, the block unfolding we are interested in is a specific case of the general block tensor unfoldings that were considered by Ragnarsson and Van Loan in [33]:

$$\mathcal{A}_{[1,\ldots,s;s+1,\ldots,d]} := \begin{bmatrix} \mathcal{A}^{1,\ldots,1,1,\ldots,1}_{(1,\ldots,s;s+1,\ldots,d)} & \cdots & \mathcal{A}^{1,\ldots,1,q_{s+1},\ldots,q_d}_{(1,\ldots,s;s+1,\ldots,d)} \\ \vdots & & \vdots \\ \mathcal{A}^{q_1,\ldots,q_s,1,\ldots,1}_{(1,\ldots,s;s+1,\ldots,d)} & \cdots & \mathcal{A}^{q_1,\ldots,q_s,q_{s+1},\ldots,q_d}_{(1,\ldots,s;s+1,\ldots,d)} \end{bmatrix}. \tag{4.6}$$

Before proceeding, consider the following illustrative examples. We will employ the well-known MATLAB notation for indexing parts of a tensor. Consider the tensor $\mathcal{A} \in \mathbb{R}^{4\times 3\times 3}$ whose slices are given by

$$\mathcal{A}(:,:,1) = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}, \quad \mathcal{A}(:,:,2) = \begin{bmatrix} 13 & 17 & 21 \\ 14 & 18 & 22 \\ 15 & 19 & 23 \\ 16 & 20 & 24 \end{bmatrix}, \text{ and } \quad \mathcal{A}(:,:,3) = \begin{bmatrix} 25 & 29 & 33 \\ 26 & 30 & 34 \\ 27 & 31 & 35 \\ 28 & 32 & 36 \end{bmatrix};$$

the vectorization of this tensor is simply the vector of integers from 1 to 36. If we would consider a general subdivision into $2 \times 2 \times 2$ subtensors, then none of the block unfoldings would coincide with any of the implicit unfoldings. Indeed, the implicit unfolding $\mathcal{A}_{(1;2,3)} = [\, \mathcal{A}(:,:,1)\ \mathcal{A}(:,:,2)\ \mathcal{A}(:,:,3) \,]$, while the block unfolding as in (4.6) would be

$$\mathcal{A}_{[1;2,3]} = \left[\begin{array}{c|c|c|c} \mathcal{A}^{1,1,1}_{(1;2,3)} & \mathcal{A}^{1,2,1}_{(1;2,3)} & \mathcal{A}^{1,1,2}_{(1;2,3)} & \mathcal{A}^{1,2,2}_{(1;2,3)} \\ \hline \mathcal{A}^{2,1,1}_{(1;2,3)} & \mathcal{A}^{2,2,1}_{(1;2,3)} & \mathcal{A}^{2,1,2}_{(1;2,3)} & \mathcal{A}^{2,2,2}_{(1;2,3)} \end{array}\right] = \left[\begin{array}{cc|cc|cc|cc|c} 1 & 5 & 13 & 17 & 9 & 21 & 25 & 29 & 33 \\ 2 & 6 & 14 & 18 & 10 & 22 & 26 & 30 & 34 \\ \hline 3 & 7 & 15 & 19 & 11 & 23 & 27 & 31 & 35 \\ 4 & 8 & 16 & 20 & 12 & 24 & 28 & 32 & 36 \end{array}\right].$$

Note that an implicit unfolding would be obtained by a suitable permutation of the columns of the block unfolding in this example. It can be verified that the block unfolding $\mathcal{A}_{[1,2;3]}$ also does not coincide with the implicit unfolding $\mathcal{A}_{(1,2;3)}$. However, if we divide $\mathcal{A}$ into subtensors of size, e.g., $3 \times 3 \times 2$, then the block unfolding becomes

$$\mathcal{A}_{[1;2,3]} = \left[\begin{array}{c|c} \mathcal{A}^{1,1,1}_{(1;2,3)} & \mathcal{A}^{1,1,2}_{(1;2,3)} \\ \hline \mathcal{A}^{2,1,1}_{(1;2,3)} & \mathcal{A}^{2,1,2}_{(1;2,3)} \end{array}\right] = \left[\begin{array}{cccccc|ccc} 1 & 5 & 9 & 13 & 17 & 21 & 25 & 29 & 33 \\ 2 & 6 & 10 & 14 & 18 & 22 & 26 & 30 & 34 \\ 3 & 7 & 11 & 15 & 19 & 23 & 27 & 31 & 35 \\ \hline 4 & 8 & 12 & 16 & 20 & 24 & 28 & 32 & 36 \end{array}\right],$$

which does coincide with the implicit unfolding $\mathcal{A}_{(1;2,3)}$. This shows that sometimes a clever choice of the division into subtensors is possible so that we can immediately

[5] See [33, Figure 1.1] for a nice illustration of this.

apply Algorithm 2 to the individual subtensors, *without first having to copy the subtensor into consecutive memory positions*, which, as we recall, is a prerequisite for applying the aforementioned algorithm.

We derived the following sufficient condition on the dimensions of the subtensors that guarantees that one block unfolding will coincide with one corresponding implicit unfolding. For the sake of unambiguity, the theorem is stated in the general case where fringe blocks may arise.

THEOREM 4.1. *Let $\mathcal{A}$ be divided into subtensors as follows:*

$$\mathcal{A} = [\mathcal{A}^{i_1,i_2,\ldots,i_d}]_{i_1,i_2,\ldots,i_d=1}^{q_1,q_2,\ldots,q_d} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_d},$$

where

$$\mathcal{A}^{i_1,i_2,\ldots,i_d} \in \mathbb{R}^{z_1\times z_2\times\cdots\times z_d}, \quad \text{with } z_j = \begin{cases} b_j & \text{if } i_j = 1,\ldots,q_j-1,\\ n_j - b_j(q_j-1) & \text{if } i_j = q_j,\end{cases}$$

and where the block sizes additionally satisfy, for a fixed choice of $1 \le s < \ell \le d$,

$$b_s \le n_s,\ b_\ell \le n_\ell, \quad \text{and} \quad b_j = \begin{cases} n_j & \text{if } j = 1,\ldots,s-1,s+1,\ldots,\ell-1,\\ 1 & \text{if } j = \ell+1,\ldots,d.\end{cases}$$

Then,

$$\mathcal{A}_{[1,\ldots,s;s+1,\ldots,d]} = \mathcal{A}_{(1,\ldots,s;s+1,\ldots,d)};$$

i.e., the block unfolding is an implicit unfolding.

Proof. We first prove the assertion for rank-1 tensors. Let $\mathcal{A} = \mathbf{a}^1 \otimes \cdots \otimes \mathbf{a}^d$, where we partition

$$\mathbf{a}^k = \left[(\mathbf{a}_1^k)^T \ \cdots \ (\mathbf{a}_{q_k}^k)^T\right]^T \text{ with } \mathbf{a}_j^k \in \mathbb{R}^{z_k},\ j = 1,2,\ldots,q_k,$$

for $k = s$ and $k = \ell$. For all other values of k, we have either $q_k = 1$ so that the partition is trivial or $q_k = n_k$ so that we can simply write a_j^k to index the jth partition of $\mathbf{a}^k$. Consider the block column of the block unfolding $\mathcal{A}_{[1,\ldots,s;s+1,\ldots,d]}$ that can be indexed by $(1,1,\ldots,1) \le (j_\ell, j_{\ell+1},\ldots,j_d) \le (q_\ell, n_{\ell+1},\ldots,n_d)$, where the inequalities should be interpreted componentwise; it is given by the matrix with block rows

$$\left[\mathcal{A}_{(1,\ldots,s;s+1,\ldots,d)}^{1,\ldots,1,j_s,1,\ldots,1,j_\ell,\ldots,j_d}\right]_{j_s=1}^{q_s}. \tag{4.7}$$

By the definition in (4.2),

$$\begin{aligned}
&\mathcal{A}_{(1,\ldots,s;s+1,\ldots,d)}^{1,\ldots,1,j_s,1,\ldots,1,j_\ell,\ldots,j_d}\\
&\quad= \left[(I,\ldots,I,S_{j_s},I,\ldots,I,S_{j_\ell},\mathbf{e}_{j_{\ell+1}},\ldots,\mathbf{e}_{j_d})^T\cdot\mathcal{A}\right]_{(1,\ldots,s;s+1,\ldots,d)}\\
&\quad= \left[\mathbf{a}^1\otimes\cdots\otimes\mathbf{a}^{s-1}\otimes\mathbf{a}_{j_s}^s\otimes\mathbf{a}^{s+1}\otimes\cdots\otimes\mathbf{a}^{\ell-1}\otimes\mathbf{a}_{j_\ell}^\ell\otimes(a_{j_{\ell+1}}^{\ell+1}\cdots a_{j_d}^d)\right]_{(1,\ldots,s;s+1,\ldots,d)}\\
&\quad= a_{j_{\ell+1}}^{\ell+1}\cdots a_{j_d}^d\cdot(\mathbf{a}^1\odot\cdots\odot\mathbf{a}^{s-1}\odot\mathbf{a}_{j_s}^s)(\mathbf{a}^{s+1}\odot\cdots\odot\mathbf{a}^{\ell-1}\odot\mathbf{a}_{j_\ell}^\ell)^T,
\end{aligned}$$

so that, substituting the block rows in (4.7) for the last expression, it follows that

$$\begin{aligned}
(4.7) &= a_{j_{\ell+1}}^{\ell+1}\cdots a_{j_d}^d\begin{bmatrix}(\mathbf{a}^1\odot\cdots\odot\mathbf{a}^{s-1})\odot\mathbf{a}_1^s\\(\mathbf{a}^1\odot\cdots\odot\mathbf{a}^{s-1})\odot\mathbf{a}_2^s\\\vdots\\(\mathbf{a}^1\odot\cdots\odot\mathbf{a}^{s-1})\odot\mathbf{a}_{q_s}^s\end{bmatrix}(\mathbf{a}^{s+1}\odot\cdots\odot\mathbf{a}^{\ell-1}\odot\mathbf{a}_{j_\ell}^\ell)^T\\
&= a_{j_{\ell+1}}^{\ell+1}\cdots a_{j_d}^d(\mathbf{a}^1\odot\cdots\odot\mathbf{a}^s)(\mathbf{a}^{s+1}\odot\cdots\odot\mathbf{a}^{\ell-1}\odot\mathbf{a}_{j_\ell}^\ell)^T,
\end{aligned}$$

where we exploited the elementary property that $\mathbf{a} \odot \left[\begin{smallmatrix}\mathbf{b}\\ \mathbf{c}\end{smallmatrix}\right] = \left[\begin{smallmatrix}\mathbf{a}\odot\mathbf{b}\\ \mathbf{a}\odot\mathbf{c}\end{smallmatrix}\right]$. We can thus write the entire block unfolding as

$$\mathcal{A}_{[1,\ldots,s;s+1,\ldots,d]} = (\mathbf{a}^1 \odot \cdots \odot \mathbf{a}^s)\left[a_{j_{\ell+1}}^{\ell+1} \cdots a_{j_d}^d (\mathbf{a}^{s+1} \odot \cdots \odot \mathbf{a}^{\ell-1} \odot \mathbf{a}_{j_\ell}^\ell)^T\right]_{j_\ell, j_{\ell+1}, \ldots, j_d = 1}^{q_\ell, n_{\ell+1}, \ldots, n_d}.$$

For every fixed choice of $(1,\ldots,1) \leq (j_{\ell+1},\ldots,j_d) \leq (n_{\ell+1},\ldots,n_d)$, the q_ℓ consecutive block columns in the above block unfolding can be written as

$$\begin{bmatrix} \mathbf{a}^{s+1} \odot \cdots \odot \mathbf{a}^{\ell-1} \odot \mathbf{a}_1^\ell \odot (a_{j_{\ell+1}}^{\ell+1} \cdots a_{j_d}^d) \\ \mathbf{a}^{s+1} \odot \cdots \odot \mathbf{a}^{\ell-1} \odot \mathbf{a}_2^\ell \odot (a_{j_{\ell+1}}^{\ell+1} \cdots a_{j_d}^d) \\ \vdots \\ \mathbf{a}^{s+1} \odot \cdots \odot \mathbf{a}^{\ell-1} \odot \mathbf{a}_{q_\ell}^\ell \odot (a_{j_{\ell+1}}^{\ell+1} \cdots a_{j_d}^d) \end{bmatrix}^T = (a_{j_{\ell+1}}^{\ell+1} \cdots a_{j_d}^d)(\mathbf{a}^{s+1} \odot \cdots \odot \mathbf{a}^{\ell-1} \odot \mathbf{a}^\ell)^T,$$

where we exploited the aforementioned elementary property again.[6] One observes that

$$\mathcal{A}_{[1,\ldots,s;s+1,\ldots,d]} = (\mathbf{a}^1 \odot \cdots \odot \mathbf{a}^s)\left[\left(\mathbf{a}^{s+1} \odot \cdots \odot \mathbf{a}^{\ell+k-1} \odot (a_{j_{\ell+k}}^{\ell+k} \cdots a_{j_d}^d)\right)^T\right]_{j_{\ell+k},\ldots,j_d=1}^{n_{\ell+k},\ldots,n_d}$$

holds for $k+1$ if it holds for $k \geq 1$; the inductive step from k to $k+1$ is, in fact, elementary because, upon close inspection, it simply repeats the definition of the Khatri–Rao product; the base case $k = 1$ was already proved above. We thus find

$$\mathcal{A}_{[1,\ldots,s;s+1,\ldots,d]} = (\mathbf{a}^1 \odot \cdots \odot \mathbf{a}^s)(\mathbf{a}^{s+1} \odot \cdots \odot \mathbf{a}^d)^T = \mathcal{A}_{(1,\ldots,s;s+1,\ldots,d)},$$

proving the rank-1 case. The general case then follows from the foregoing discussion and linearity, i.e., from the fact that every tensor can be written as a linear combination of rank-1 tensors (e.g., as in (2.1)). ∎

The theorem states that if $\mathcal{A}$ is divided into subtensors of a suitable size, then *the block unfolding* (4.6) *is an implicit unfolding*, requiring, hence, no explicit reorganization of the data elements when computing (3.3) at the block level.

4.3. Automatically selecting the block size. For a general-purpose blockwise algorithm, the dimensions of the subtensors should be chosen automatically. We propose the following heuristic, which allows the user to specify the maximum number of elements in the subtensor: $b_1 \cdots b_d \leq c$. The suggested algorithm will produce a division into subtensors of size

$$n_1 \times \cdots \times n_{s-1} \times b_s \times n_{s+1} \times \cdots \times n_{\ell-1} \times b_\ell \times 1 \times \cdots \times 1,$$

with $1 \leq s < \ell \leq d$.[7] Consequently, if we take s as the splitting point for the implicit unfolding, it follows that the produced subtensor division is compatible with implicit block unfoldings. From the complexity analysis in section 4.1, it follows that the subtensor shape that minimizes both the number of floating-point operations as well as the additional memory consumption is such that

$$b_1 \cdots b_s \approx b_{s+1} \cdots b_d, \text{ subject to } 1 \leq b_i \leq n_i,\ i = 1,\ldots,d.$$

[6] If $s+1 = \ell$, then the above equality is elementary in itself.

[7] Smaller subtensor sizes may occur at the boundary if the division is not perfect.

We determine an initial splitting point s', satisfying

$$s' = \arg\min_{1\le j\le d} \frac{\max\{n_1\cdots n_j, n_{j+1}\cdots n_d\}}{\min\{n_1\cdots n_j, n_{j+1}\cdots n_d\}}.$$

This optimum can be computed by trying at most all d possibilities. Let $\tau_1 \leftarrow \sqrt{c/2}$ be the number of rows of the implicitly unfolded subtensor that we hope to achieve. Then, setting $i \leftarrow 1$, we repeatedly compute

$$b_i \leftarrow \begin{cases} n_i & \text{if } n_i \le 2\tau_i, \\ \lfloor \tau_i \rfloor & \text{otherwise,} \end{cases} \quad \text{and, thereafter,} \quad \tau_{i+1} \leftarrow \tau_i / b_i \text{ and } i \leftarrow i+1. \tag{4.8}$$

This process is stopped at the beginning of step i if i equals $s'+1$ or $\tau_i < 2$. In either case, let $s \le s'$ be one less than the value of i at which point the updating process was stopped. Then, we set the target number of columns, $\tau_{s+1} \leftarrow \min\{2, \tau_{s+1}\} \cdot \sqrt{c/2}$. Subsequently, the block sizes are determined as in (4.8). The process stops at the beginning of step i if either $i = d+1$ or $\tau_i < 2$. The other block sizes are set to 1. We will refer to this approach as the automatic block selection (ABS) algorithm.

We suggest choosing the parameters $\{t_k\}_{k=0}^{\gamma}$ required in the division into a sequence of γ CPGs in (4.5) by dividing the sequence into chunks of approximately equal length. Let $t_0 = 0$, and choose

$$\gamma \leftarrow \left\lceil \frac{r}{r_0} \right\rceil, \text{ and then } t_k \leftarrow t_{k-1} + \left\lfloor \frac{r}{\gamma} \right\rfloor + \left[k \le r - \gamma \left\lfloor \frac{r}{\gamma} \right\rfloor \right],$$

where $[a \le b]$ equals one if $a \le b$ and zero otherwise. This choice ensures that $0 \le t_k - t_{k+1} - \lfloor \frac{r}{\gamma} \rfloor \le 1$, so that the division is as equal as possible. This heuristic additionally ensures that the minimum number of vectors involved in a single CPG is sufficiently large:

$$r_0 \ge t_k - t_{k-1} \ge \left\lfloor \frac{r}{\gamma} \right\rfloor = \left\lfloor \frac{\gamma-1}{\gamma} r_0 + \frac{\delta}{\gamma} \right\rfloor \ge \left\lfloor \frac{\gamma-1}{\gamma} r_0 \right\rfloor,$$

where $\delta \in [1, r_0]$ such that $r = (\gamma-1)r_0 + \delta$; it will be shown in the numerical experiments in section 6.2 why this is an important property.

Assume that the ABS algorithm stopped repeating (4.8) when $\tau_i < 2$ when determining both the number of rows as well as the number of columns of the implicitly unfolded tensor. Then, it produces block sizes satisfying

$$\tfrac{1}{2\sqrt{2}}\sqrt{c} < b_1\cdots b_s \le \sqrt{2}\sqrt{c},\ \tfrac{1}{4\sqrt{2}}\sqrt{c} < b_{s+1}\cdots b_d < 2\sqrt{2}\sqrt{c},\ \text{and } \tfrac{1}{4}c < b_1\cdots b_d \le c;$$

otherwise, only the upper bounds in the above ranges apply. It follows from these bounds that the memory consumption for the proposed blocked implementation is

$$\mathcal{O}(3\sqrt{2}\sqrt{c}r_0).$$

In practice, this constant can be chosen sufficiently small to substantially improve the memory consumption with regard to the standard unblocked algorithm in [31].

In all of the experiments presented in section 6, the block size for the blocked implementations and the division of the number of vectors were determined automatically using the heuristics explained above.

5. Implementation details. The algorithms were implemented in C++ using the matrix library Eigen3 [18], which is a header-only C++ library offering matrix data structures and corresponding algorithms. For reasons of performance, we compiled the OpenBLAS [1] implementation of the BLAS interface, which extends GotoBLAS [17] to some newer architectures. Our preliminary experiments indicated that better throughput could be achieved with the matrix multiplication provided by OpenBLAS: it attained up to 95% of the peak performance in our experimental setup, while Eigen only achieved up to 85%. All matrix products are computed by calling the `dgemm` routine of OpenBLAS.

Temporary storage. For avoiding expensive memory allocations and deallocations during the computation of the CPG, we allocate three arrays of double-precision floating-point numbers, each holding $r_0 \cdot \max\{b_1 \cdots b_s, b_{s+1} \cdots b_d\}$ values; *the maximum memory consumed by our blockwise implementation is thus* $6\sqrt{2cr_0}$ *memory items*, or $48\sqrt{2cr_0}$ bytes for double precision floating-point numbers. One of the arrays is used to temporarily store B or C as it appears on lines 1 and 11 of Algorithm 2. The other two arrays are used in the computation of the Khatri–Rao product that is required at the same place, as well as in the computation of the right-to-left contraction in lines 8 and 18. The result of one step of the left-to-right contraction in lines 6 and 16 is, essentially, stored in the first array.

Computation of Khatri–Rao product. A Khatri–Rao product $A \odot B$ is formed by computing $\mathbf{b}_i \otimes_K \mathbf{a}_i$ for every column i. Practically, we compute the result columnwise as the column-major vectorization of $\mathbf{a}_i \mathbf{b}_i^T$. For computing, e.g., $W := V_1 \odot \cdots \odot V_s$, we proceed as follows. $W_1 := V_1 \odot V_2$ is computed as above, and the result is stored in the first temporary array. Then, we compute $W_2 := W_1 \odot V_3$, storing the result in the second array. We then swap the pointers of the two temporary arrays and proceed with computing the next Khatri–Rao product. Computing the Khatri–Rao product reduces to computing several outer products, an operation that attains only a very low throughput. For instance, in our experimental setup, the Khatri–Rao products could in the most favorable circumstances attain only up to 10% of the peak performance. For reasons of performance, it is important to handle a fringe case explicitly, namely when at least one of the block sizes equals 1. Let the sequence P be defined as $P = \langle i \mid V_i \in \mathbb{R}^{1 \times r} \rangle$, and consider Algorithm 2. In lines 1 and 11, all factors in P should be removed from the Khatri–Rao product. Then, in lines 6 and 16, the vector should be replaced with 1 if $k \in P$, i.e., nothing should be computed. In lines 8 and 18, the right-to-left contraction can also skip all vectors $\mathbf{v}_j^{(i)}$ with $j \in P$. Thereafter, in the same lines, $\mathbf{w}_k^{(i)}$ should be multiplied with $\prod_{j \in P} \mathbf{v}_j^{(i)}$. This multiplication is well defined, because these vectors are just scalars.

6. Numerical experiments. In this section, we compare the performance of the proposed blockwise algorithm with the standard Algorithm 2 of [31]. For ensuring optimal testing conditions, the compiled executable was started with `numactl --physcpubind=+0 -l` to pin its execution on the first physical processing core. OpenBLAS was instructed to use only one processing unit by calling the function `openblas_set_num_threads(1)`. In addition, after allocating all memory our program requires, an `mlock` system call was made from the code, requesting that the current memory pages used by the executable be retained in the system's main memory during its execution. The code was compiled with the flags `-O3`, `-std=c++0x`, `-msse4`, `-fwhole-program`, `-funroll-loops`, and `-malign-double` using the GCC v4.7.1. Only one computational thread was used in all experiments.

The computer system on which the experiments were performed consisted of one

Intel Xeon X5550 quad-core processor clocked at 2.67GHz, 8MB L2 cache memory, and 16GB of main memory. Because the Intel Turbo Boost technology can increase the clock speed of an individual core up to 3.06GHz if the others are unloaded, the peak performance with a single computational thread was 12.24 Gflop/s: it can concurrently complete two double-precision floating-point additions and two multiplications.

Generating random tensors. We will compare the performance of the implementations on a large number of random tensors with various shapes. The results will then be aggregated so as to give indications of the performance that may be expected. We note that the performance of the proposed algorithm for computing CPGs depends only on the shape $(n_1, \ldots, n_d)$ of the tensor and on the number of vectors r. In particular, the performance depends neither on the true rank of the tensor nor on the specific numerical values of the entries of the tensor. That is, the performance of our algorithm will be independent of any possible structure that is present in the tensor. For this reason and for the sake of simplicity, we will generate tensors whose entries are random double-precision floating-point numbers uniformly distributed between 0 and 1.[8] An alternative approach consists of generating r-term CP decompositions and then adding some random Gaussian noise of small magnitude. This technique is often used for testing the performance of algorithms for computing approximate CP decompositions.

For generating random shapes, the following procedure was employed. Assume that we are given a target number of elements C, and we wish to determine a shape $(n_1, \ldots, n_d)$ such that $\frac{1}{2}C \leq n_1 \cdots n_d \leq C$. Set $n \leftarrow \lfloor (d-1)C^{1/d} \rfloor$ and $C_1 \leftarrow C$. Then, we iteratively set, for $i = 1, \ldots, d-1$,

$$n_i \leftarrow \min\{\mathrm{mod}(\zeta_i, n), C_i\} \quad \text{and} \quad C_{i+1} \leftarrow C_i / n_i,$$

where $\zeta_i \in \mathbb{N}$ are random integers. As last step we set $n_d \leftarrow \lfloor C_d \rfloor$. If one of the n_i equals zero, then the shape is discarded. This procedure will generate shapes that are mostly balanced but also allows for some factors that are much larger than the others.

As can be understood from the time and space complexity of Algorithm 2 and Algorithm 3, it is beneficial to reorganize the factors of the tensor prior to computing the CPG so that $\max\{\prod_{i=1}^{s} \frac{1}{n_i}, \prod_{i=s+1}^{d} \frac{1}{n_i}\}$ is minimized. The optimal order can in principle be determined by investigating all permutations; however, in this paper, we will assume that the factors of the tensor are reordered using the following heuristic. We sort the sizes of the factors $\langle n_1, n_2, \ldots, n_d \rangle$ by decreasing magnitude; say that $\langle n_{q_1}, n_{q_2}, \ldots, n_{q_d} \rangle$ is the resulting sequence. We assign $n_{p_1} \leftarrow n_{q_1}$ and $n_{p_d} \leftarrow n_{q_2}$ and set $l \leftarrow 2$ and $r \leftarrow d-1$. Then, for increasing values of $i = 3, \ldots, d$, we set either $n_{p_l} \leftarrow n_{q_i}$ and $l \leftarrow l+1$ if $\prod_{j=1}^{l-1} n_{p_j} < \prod_{j=r+1}^{d} n_{p_j}$, or $n_{p_r} \leftarrow n_{q_i}$ and $r \leftarrow r-1$ otherwise. The sizes of the factors of the reordered tensor are then $\langle n_{p_1}, n_{p_2}, \ldots, n_{p_d} \rangle$. This heuristic facilitates the selection of a good splitting point, and it may be expected to produce more balanced splittings than the heuristic suggested in [31], which suggests choosing a permutation $\mathbf{p}'$ so that $n_{p'_1} \leq n_{p'_2} \leq \cdots \leq n_{p'_d}$. In all of the experiments, we assume that the factors of the tensor have been reorganized following the heuristic described above. This is a reasonable assumption in optimization algorithms for computing a CP decomposition, as one would reorganize the factors of the tensor just once prior

[8] Tensors generated in this way will have a rank that is of the order $(\prod_{k=1}^{d} n_k)/(1+\sum_{k=1}^{d}(n_k-1))$ with probability 1 (see, e.g., [26]) and are not well approximated by a tensor of low rank. However, for the sake of verifying the performance of the CPG, none of these concerns are influential.

to computing the decomposition. In this way, the cost of reorganization is amortized over the cost of the execution of the optimization algorithm.

As an illustration of shapes that may be produced and appear in our experiments, we present below the first 30 shapes generated when the target size is set to $C = 250{,}000{,}000$ and the number of factors of the tensor is 3:

$(6590, 116, 327)$	$(848, 484, 609)$	$(1023, 440, 555)$	$(1036, 350, 688)$	$(2801, 210, 425)$
$(879, 512, 555)$	$(4948, 173, 292)$	$(8142, 42, 731)$	$(1855, 217, 621)$	$(914, 325, 841)$
$(775, 420, 767)$	$(1242, 212, 947)$	$(1688, 213, 695)$	$(969, 484, 533)$	$(1032, 304, 796)$
$(1196, 205, 1015)$	$(976, 457, 560)$	$(1082, 354, 652)$	$(1249, 204, 981)$	$(1200, 176, 1183)$
$(1071, 269, 865)$	$(683, 591, 619)$	$(923, 499, 542)$	$(847, 373, 791)$	$(5753, 102, 426)$
$(400641, 6, 104)$	$(1197, 356, 586)$	$(909, 454, 605)$	$(5688, 187, 235)$	$(1810, 229, 603)$

6.1. Blockwise versus standard computation without sequencing. We investigate and compare the performance of the blockwise and standard algorithms for several choices of the parameter c in the ABS algorithm. Here, we consider $r_0 = \infty$. Based on these results, we propose a good choice for r_0, and then we investigate the performance for $r > r_0$ in the next subsection.

We generated 200 random tensors with $d = 3, 4, 5$ factors and $C = 250{,}000{,}000$ using the algorithm outlined above. This results in tensors consuming between 1GB and 2GB of memory. For every shape and every $r = 50$, 100, 150, 250, 400, 650 in the r-term CP decomposition, we measured the total execution time for computing three CPGs with $\{V_j \in \mathbb{R}^{n_j \times r}\}_{j=1}^{d}$ with independent and identically distributed (i.i.d.) standard normally distributed elements using the blocked algorithm for $c = 2^{19}$, 2^{21}, 2^{23}, and with the standard algorithm.

Figure 1 summarizes the relative execution time of the blocked algorithm with respect to the unblocked algorithm, i.e., the execution time of the blocked algorithm divided by the execution time of the unblocked algorithm expressed as a percentage; a lower value is better for the blockwise algorithm. As a first observation, we note that for at least 95% of the tested shapes, the blocked algorithm $c = 2^{19}$, corresponding to a memory consumption of $48r$ KB, is no more than 11% slower than the unblocked algorithm, with $c = 2^{21}$ it is no more than 5.5% slower, and with $c = 2^{23}$ it is only 4% slower. The performance gap increases modestly with r; this effect is due to the fact that the throughput of a matrix product generally increases with the dimensions of the matrix. It may be expected that if r is sufficiently large, then the observed throughput will no longer increase as r increases. This can be observed in Figure 2, where we plot the performance of the two methods with respect to the theoretical peak performance of the computer system; higher is better. As can be seen, for $r \geq 250$ and $c = 2^{21}$ and 2^{23}, the median throughput of the blockwise algorithm is more than 90% of the peak performance of the machine. Note that the performance for $c = 2^{19}$ is significantly worse than the other two parameter choices. This is because we are essentially multiplying two matrices whose sizes are of the order $\sqrt{c} \times \sqrt{c}$ and $\sqrt{c} \times r$. For the three values of c tested, we have $\sqrt{c} \approx 724, 1448$, and 2896, respectively. The unblocked algorithm, on the other hand, will compute matrix products with matrices whose expected dimensions are of the order $\sqrt{C} \times \sqrt{C}$ by $\sqrt{C} \times r$, with $\sqrt{C} \approx 15811$. Our experiments indicate that with OpenBLAS, there is little performance difference between multiplying a 1500×1500 with a $1500 \times r$ matrix and multiplying a 15000×15000 with a $15000 \times r$ matrix. This 100-fold increase in size only results in a 4% increase in throughput. Consequently, the blocked algorithm can be competitive in execution time while multiplying a set of much smaller matrices than the unblocked algorithm.

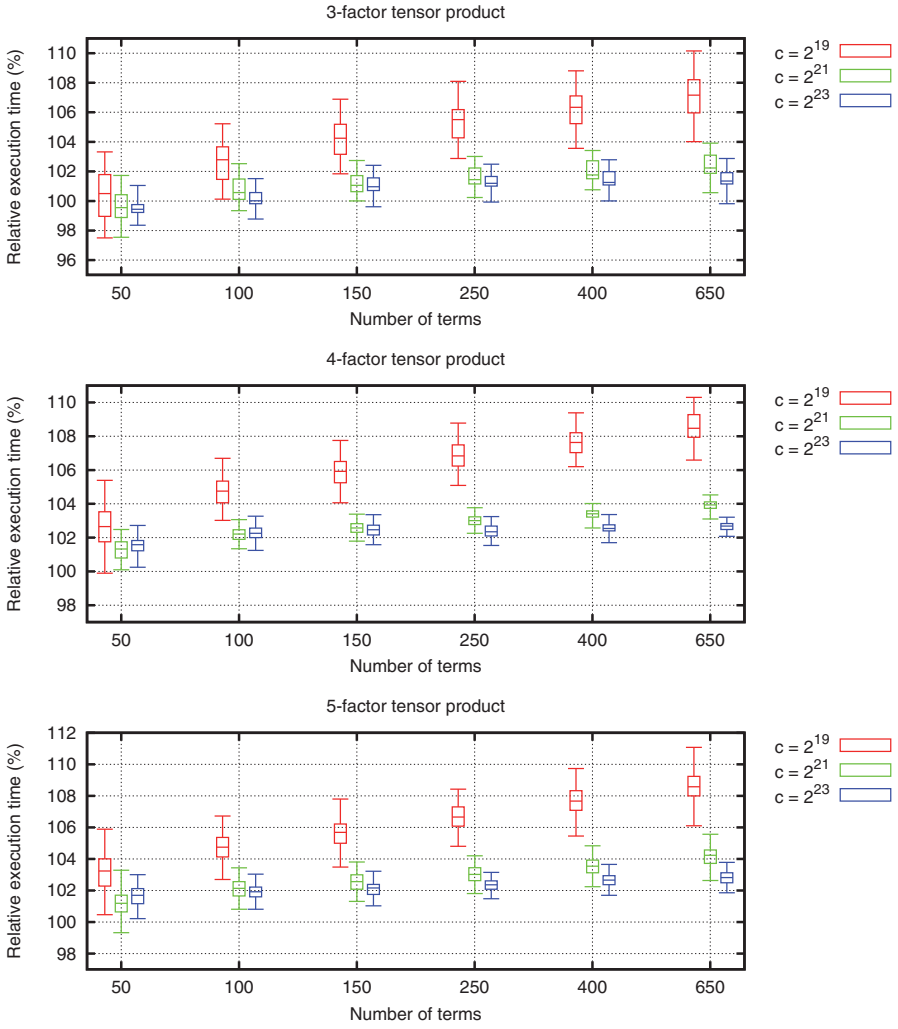

FIG. 1. *Visualization of the relative execution time of the blocked algorithm with respect to the unblocked algorithm. A single box plot summarizes the experimental results over all* 200 *tested shapes for a combination of the number of factors, the number of terms* r*, and the parameter* c *of the ABS algorithm. The whiskers of the box plots enclose* 95% *of all experimental trials.*

Finally, we demonstrate that the blocked implementation is effective at reducing the memory requirements. In Figure 3 we plot the fraction of memory required by the unblocked algorithm relative to the blocked algorithm; higher values are better for the latter. Note that the memory requirements are linear in r, because we consider $r_0 = \infty$ here, so that there is no dependency on r when comparing the relative memory consumptions. From the figure, one learns that in over half of the tested cases the unblocked algorithm required one order of magnitude more memory than the blocked algorithm. By choosing the parameter c of the ABS algorithm, the memory requirements can be influenced. For $c = 2^{21}$, the median improvement was a factor of 20, and with $c = 2^{19}$ it was 30. The improvement of the blocked algorithm is largest for 3-factor tensor spaces, as could have been anticipated from the asymptotic

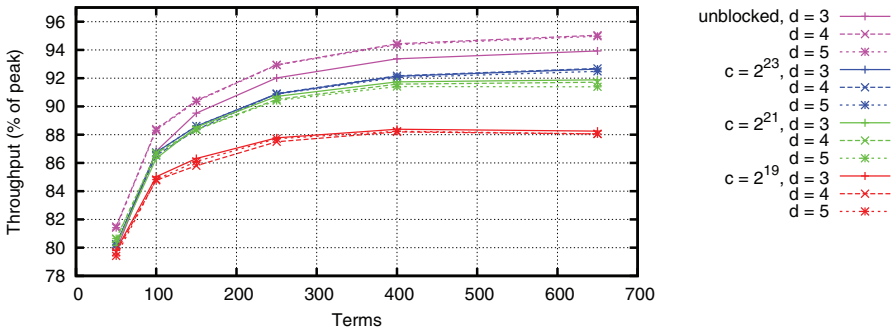

FIG. 2. *A comparison of the unblocked algorithm and the blocked algorithm. The median over the* 200 *tested shapes of the throughput, relative to the theoretical peak performance of the computer system, is plotted for several values of the parameter* c *of the ABS algorithm and for the number of factors* d*.*

space complexity. In this case, the median improvement was a factor of 50. Setting $c = 2^{19}$, the improvement was at least a factor of 200 in 75% of all tested tensor shapes; naturally, this comes at the cost of an increased execution time, as we showed in Figure 1.

6.2. Blockwise versus standard computation with sequencing. From Figures 1 and 2, we can conclude that for a sufficiently large parameter choice of c in the ABS algorithms, e.g., $c = 2^{21}$ or 2^{23}, and a small number of terms, i.e., $r \leq 650$, the proposed blockwise algorithm will not be more than 6% slower than the standard algorithm for at least 95% of the tested shapes. As the absolute performance of both methods is excellent for a large number of terms, i.e., higher than 90% of the peak performance of the computer system, it may be expected that for a higher number of terms, the proposed blockwise algorithm cannot be more than 11% slower than the standard algorithm. By sequencing the CPG into multiple CPGs, it may be expected that we can attain at least 90% of the peak performance if we choose r_0 sufficiently large.

Note that the proposed heuristic for sequencing in section 4.3 ensures that $\lfloor \frac{1}{2} r_0 \rfloor \leq \lfloor \frac{\gamma - 1}{\gamma} r_0 \rfloor \leq t_l - t_{l-1} \leq r_0$ for every l. That is, the heuristic not only ensures that the number of terms in one partial CPG is no more than r_0, for the sake of memory consumption, but it also guarantees a minimum number of terms involved in a single partial CPG. This enables good absolute performance for every $r \geq \frac{1}{2} r_0$. Indeed, say that we wish to attain at least 90% of the peak performance; then it suffices to choose c and r_0 such that for every $\frac{1}{2} r_0 \leq r \leq r_0$ the performance is at least 90%. From Figure 2, we learn that $r_0 = 500$ with c at least 2^{21} is a likely candidate, because the blockwise algorithm attains just over 90% of the peak performance at $r = 250 = \frac{1}{2} r_0$. Based on theoretical considerations, it is already clear that this choice yields the desired fraction of the peak performance if $r \geq 250 = \frac{1}{2} r_0$, employing only a constant amount of memory.

Next, we illustrate that sequencing the CPG with $r_0 = 500$ yields great performance for large r. We generated 250 random 3-factor tensors using the algorithm from section 6 with $C = 62{,}500{,}000$, resulting in tensors consuming between 250 and 500MB of memory. For every shape and number of terms $r = 600, 700, \ldots, 1500$ in the CP decomposition, we measured the execution time for computing three CPGs with random $\{V_j \in \mathbb{R}^{n_j \times r}\}_{j=1}^{d}$ using the blockwise algorithm ($c = 2^{21}, 2^{22}$) and with

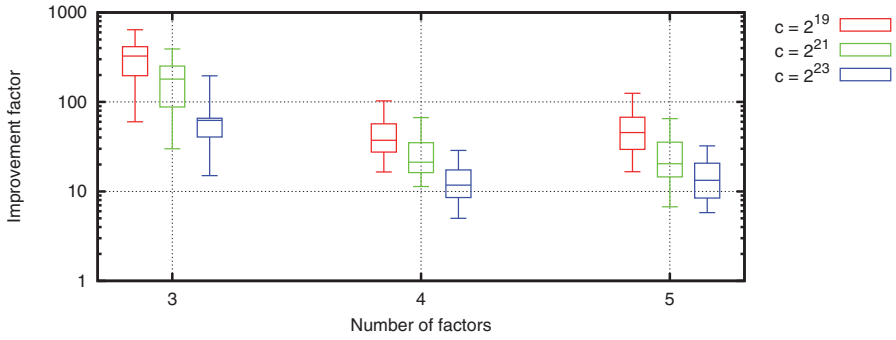

FIG. 3. *Visualization of the improvement in memory consumption of the blocked algorithm with respect to the unblocked algorithm. A single box plot summarizes the experimental results over all* 200 *tested shapes for a combination of factors of the tensor* d *and the parameter* c *in the ABS algorithm.*

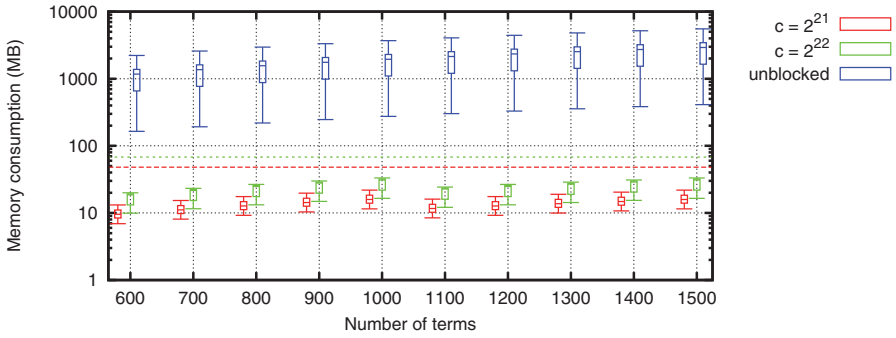

FIG. 4. *Visualization of the absolute memory consumption of the blockwise algorithm and the standard algorithm. A single box plot summarizes the experimental results over all* 250 *tested shapes for a fixed number of terms* r*. The whiskers of the box plots enclose* 95% *of all experimental trials. The dashed line at* $48MB$ *is the upper bound on the memory consumption for the blockwise algorithm with* $c = 2^{21}$*, and similarly for the dashed line at* $96MB$ *for the parameter choice* $c = 2^{22}$*.*

Algorithm 2 from [31].

The constant memory consumption of the proposed algorithm is illustrated in Figure 4. The two horizontal lines at 48MB and 96MB represent the upper bound on the memory usage of the proposed blockwise algorithm with parameter $c = 2^{21}$ and 2^{22}, respectively, of the ABS algorithm. The figure highlights the dependency on the number of terms r for the standard algorithm: the median increases from about 1GB to 4GB; from theoretical considerations, we know that the dependency is linear. For the blockwise algorithm, we note a sudden drop in memory consumption at $r = 1100$. This is because at $r = 1000$, the CPG is sequenced into two CPGs, each with 500 terms. At $r = 1100$, the CPG is sequenced into three CPGs with approximately 367 terms each.

From Figure 5, it follows that sequencing does not materially increase the execution time with respect to the standard algorithm. By combining theoretical considerations with the experimental data from Figure 1, we already anticipated that the execution time should not rise by more than 11%. As can be seen in Figure 5, in at least 75% of the tested shapes the increase was only half of that. For $r \leq 500$, Figure

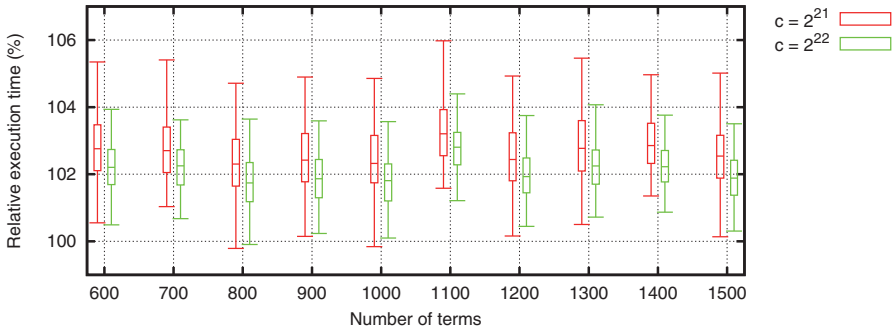

FIG. 5. *Visualization of the relative execution time of the blockwise algorithm with respect to the standard algorithm. A single box plot summarizes the experimental results over all* 250 *tested shapes for a combination of the number of terms* r *and the parameter* c *of the ABS algorithm. The whiskers of the box plots enclose* 95% *of all experimental trials.*

1 applies.

7. Conclusions. We proposed a blockwise algorithm for computing the gradient of the objective function in gradient-based optimization algorithms for constructing an r-term CP decomposition. The algorithm consumes only a constant amount of memory, improving on the state-of-the-art approach from [31]. Crucially, the savings in memory consumption do not materially affect the execution time: the proposed method attains upward of 90% of the peak performance of the computer system for large r. Dividing the tensor into subtensors is accomplished by a heuristic block selection algorithm; also, the sequencing into multiple CPG computations is performed automatically. The memory consumption of the proposed algorithm can be influenced by the user through careful selection of two parameters. In our experiments, we found that, for a sufficiently large number of terms r in an r-term CP decomposition, over 90% of the peak performance can be attained using at most 50MB of memory, corresponding to the parameter selection $c = 2^{21}$ and $r_0 = 500$. A good choice for these parameters depends intrinsically on the performance of the `dgemm` BLAS3 routine in multiplying matrices whose size is of the order $\sqrt{c} \times \sqrt{c}$ and $\sqrt{c} \times r_0$. As such, we believe that choosing $c \geq 2^{21}$ and $r_0 \geq 500$ should yield at least acceptable performance on a variety of modern computer systems.

It appears to us that the algorithm that is proposed in this paper cannot be modified straightforwardly for computing the "sequential gradient" that appears in ALS-type methods for constructing CP decompositions—contrary to the algorithm from [31]. The reason is that at the block level only a partial update to the required factor matrices is available, and it appears to us that aggregating all required updates cannot be accomplished in constant memory. Notwithstanding this limitation, we are convinced that the presented algorithm will find application in direct optimization algorithms for computing the CPG. Such algorithms have consistently been found to outperform ALS-type algorithms on difficult problems in terms of the accuracy of the recovered decomposition [3, 24, 36, 38]; in terms of computational performance, these algorithms are usually competitive with ALS-type methods. Additionally, direct optimization algorithms employing the proposed blockwise implementation of the CPG will enjoy a smaller memory consumption than the ALS-type algorithms employing a

sequential gradient whose memory consumption cannot be reduced straightforwardly with the presented blocking techniques, potentially limiting their applicability with respect to the proposed implementation.

Acknowledgments. We express gratitude to the reviewers for their very detailed comments that helped to improve the quality of this paper substantially, especially in regard to the introduction and Theorem 4.1.

N. Vanbaelen is heartily thanked for contributions to the program code while he was writing his Master's thesis on the parallelization of tensor decompositions. We thank T. Kolda for comments that substantially improved the proposed method. The first author thanks L. Sorber and A.-J. Yzelman for encouraging and delightful discussions.

REFERENCES

[1] *Openblas v*1.13, http://www.openblas.net (last accessed November 6, 2014).
[2] *TiledArray: A Massively-Parallel, Block-Sparse Tensor Library Written in C++*, https://github.com/ValeevGroup/tiledarray (last accessed November 6, 2014).
[3] E. Acar, D. M. Dunlavy, and T. G. Kolda, *A scalable optimization approach for fitting canonical tensor decompositions*, J. Chemometrics, 25 (2011), pp. 67–86.
[4] E. Acar, D. M. Dunlavy, T. G. Kolda, and M. Mørup, *Scalable tensor factorizations for incomplete data*, Chemometr. Intell. Lab., 106 (2011), pp. 41–56.
[5] C. J. Appellof and E. R. Davidson, *Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents*, Anal. Chem., 53 (1981), pp. 2053–2056.
[6] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrisson, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, *Synthesis of high-performance parallel programs for a class of* ab initio *quantum chemistry models*, Proc. IEEE, 93 (2005), pp. 276–292.
[7] C. Bocci, L. Chiantini, and G. Ottaviani, *Refined methods for the identifiability of tensors*, Ann. Mat. Pura Appl. (4), 193 (2014), pp. 1691–1702.
[8] J. Carroll and J.-J. Chang, *Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart–Young" decomposition*, Psychometrika, 35 (1970), pp. 283–319.
[9] L. Chiantini, G. Ottaviani, and N. Vannieuwenhoven, *An algorithm for generic and low-rank specific identifiability of complex tensors*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 1265–1287.
[10] V. de Silva and L.-H. Lim, *Tensor rank and the ill-posedness of the best low-rank approximation problem*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 1084–1127.
[11] H. De Sterck and M. Winlaw, *A nonlinear preconditioned conjugate gradient algorithm for rank-R canonical tensor approximation*, Numer. Linear Algebra Appl., 22 (2015), pp. 410–432.
[12] I. Domanov and L. De Lathauwer, *On the uniqueness of the canonical polyadic decomposition of third-order tensors—part* II: *Uniqueness of the overall decomposition*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 876–903.
[13] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, *A set of level* 3 *basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.
[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of Fortran basic linear algebra subprograms*, ACM Trans. Math. Software, 14 (1988), pp. 1–17.
[15] L. Eldén and B. Savas, *A Newton–Grassmann method for computing the best multilinear rank-*(r_1, r_2, r_3) *approximation of a tensor*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 248–271.
[16] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. I. Krylov, *New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations*, J. Comput. Chem., 34 (2013), pp. 2293–2309.
[17] K. Goto and R. A. van de Geijn, *Anatomy of high-performance matrix multiplication*, ACM Trans. Math. Software, 34 (2008), 12.
[18] G. Guennebaud, B. Jacob, et al., *Eigen v*3, http://eigen.tuxfamily.org (2010).

[19] S. Hansen, T. Plantenga, and T. G. Kolda, *Newton-Based Optimization for Kullback-Leiber Nonnegative Tensor Factorizations*, preprint, arXiv:1304.4964, http://arxiv.org/abs/1304.4964, 2014.

[20] R. A. Harshman, *Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis*, UCLA Working Papers in Phonetics, 16 (1970), pp. 1–84.

[21] C. Hayashi and F. Hayashi, *A new algorithm to solve PARAFAC-model*, Behaviormetrika, 11 (1982), pp. 49–60.

[22] F. L. Hitchcock, *Multiple invariants and generalized rank of a p-way matrix or tensor*, J. Math. Phys., 7 (1927), pp. 39–79.

[23] F. L. Hitchcock, *The expression of a tensor or a polyadic as a sum of products*, J. Math. Phys., 6 (1927), pp. 164–189.

[24] P. K. Hopke, P. Paatero, H. Jia, R. T. Ross, and R. A. Harshman, *Three-way (PARAFAC) factor analysis: Examination and comparison of alternative computational methods as applied to ill-conditioned data*, Chemometr. Intell. Lab., 43 (1998), pp. 25–42.

[25] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51 (2009), pp. 455–500.

[26] J. M. Landsberg, *Tensors: Geometry and Applications*, Grad. Stud. Math. 128, AMS, Providence, RI, 2012.

[27] L. S. Lorente, J. M. Vega, and A. Velazquez, *Compression of aerodynamic databases using high-order singular value decomposition*, Aerosp. Sci. Technol., 14 (2010), pp. 168–177.

[28] I. V. Oseledets and D. V. Savost'yanov, *Minimization methods for approximating tensors and their comparison*, Comp. Math. Math. Phys., 46 (2006), pp. 1641–1650.

[29] P. Paatero, *A weighted non-negative least squares algorithm for three-way 'PARAFAC' factor analysis*, Chemometr. Intell. Lab., 38 (1997), pp. 223–242.

[30] A.-H. Phan and A. Cichocki, *PARAFAC algorithms for large-scale problems*, Neurocomputing, 74 (2011), pp. 1970–1984.

[31] A.-H. Phan, P. Tichavský, and A. Cichocki, *Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations*, IEEE Trans. Signal Process., 61 (2013), pp. 4834–4846.

[32] A.-H. Phan, P. Tichavský, and A. Cichocki, *Low complexity damped Gauss–Newton algorithms for CANDECOMP/PARAFAC*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 126–147.

[33] S. Ragnarsson and C. F. Van Loan, *Block tensor unfoldings*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 149–169.

[34] S. Ragnarsson and C. F. Van Loan, *Block tensors and symmetric embeddings*, Linear Algebra Appl., 438 (2013), pp. 853–874.

[35] M. D. Schatz, T. M. Low, R. A. van de Geijn, and T. G. Kolda, *Exploiting symmetry in tensors for high performance: Multiplication with symmetric tensors*, SIAM J. Sci. Comput., 36 (2014), pp. C453–C479.

[36] L. Sorber, M. Van Barel, and L. De Lathauwer, *Optimization-based algorithms for tensor decompositions: Canonical polyadic decomposition, decomposition in rank-$(l_r, l_r, 1)$ terms, and a new generalization*, SIAM J. Optim., 23 (2013), pp. 695–720.

[37] G. Tomasi and R. Bro, *PARAFAC and missing values*, Chemometr. Intell. Lab., 75 (2005), pp. 163–180.

[38] G. Tomasi and R. Bro, *A comparison of algorithms for fitting the PARAFAC model*, Comput. Statist. Data Anal., 50 (2006), pp. 1700–1734.

[39] N. Vannieuwenhoven, N. Vanbaelen, K. Meerbergen, and R. Vandebril, *The Dense Multiple-Vector Tensor-Vector Product: An Initial Study*, Tech. Report TW635, KU Leuven, Leuven, Belgium, 2013.